Docket No. 200311046-1

10/601,992

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR U.S. LETTERS PATENT

Title:

COST-AWARE ADMISSION CONTROL FOR STREAMING MEDIA SERVER

Inventors:

Ludmila Cherkasova
1338 Elsona Drive
Sunnyvale, CA 94087

Wenting Tang
666 Gail Avenue, Apt. C25
Sunnyvale, CA 94086
Citizenship: People's Republic of China

COST-AWARE ADMISSION CONTROL FOR STREAMING MEDIA SERVER

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is related to co-pending and commonly assigned U.S. Patent Application Serial Number 10/306,279 filed November 27, 2002 entitled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER," the disclosure of which is hereby incorporated herein by reference.

FIELD OF THE INVENTION

[0002] The present invention relates in general to admission control strategies and more specifically to systems and methods for managing the admission of requests for streaming media files to a streaming media server.

DESCRIPTION OF RELATED ART

[0003] Today, much information is stored as digital data that is available to processor-based devices via client-server networks. Client-server networks are delivering a large array of information (including content and services) such as news, entertainment, personal shopping, airline reservations, rental car reservations, hotel reservations, on-line auctions, on-line banking, stock market trading, as well as many other services and types of content. Such information providers (sometimes referred to as "content providers") are making an increasing amount of information available to users via client-server networks.

[0004] An abundance of information is available on client-server networks, such as the Internet, Intranets, the World Wide Web (the "web"), other Wide and Local Area Networks (WANs and LANs), wireless networks, and combinations thereof, as examples, and the amount of information available on such client-server networks is continuously increasing. Further, users are increasingly gaining access to client-server networks, such as the web, and commonly look to such client-server networks (as opposed to or in addition to other sources of information) for desired information. For example, a relatively large segment of the human population has access to the Internet via personal computers (PCs), and Internet access is now possible with many mobile devices, such as personal digital assistants (PDAs), mobile telephones (e.g., cellular telephones), etc.

[0005] An increasingly popular type of technology for providing information to clients is known as "streaming media." Streaming media is a well-known technology in the computer arts. In general, streaming media presents data (e.g., typically audio and/or video) to a client in a streaming or continuous fashion. That is, with streaming media a client is not required to receive all of the information to be presented before the presentation begins. Rather, presentation of information in a streaming media file may begin before all of the file is received by the client, and as the received portion of the file is being presented, further portions of the file continue to be received by the client for later presentation. Thus, streaming media involves media (e.g., typically audio and/or video) that is transmitted from a server (a media server) to a client and begins playing on the client before fully downloaded.

[0006] Streaming media is a particularly popular technique for communicating audio and/or video files from a server to a client. Audio and video files tend to be quite large, even after being compressed. If streaming media is not used, an entire file is generally required to be downloaded to a client before the client can begin to play the file. Such a download may require an undesirably long delay before the client can begin playing the file. With streaming media (e.g., streaming audio or streaming video), a client is not required to wait until the entire file is downloaded to play it. Instead, the client can begin playing the file (e.g., presenting the video and/or audio to a user) while it downloads to the client.

[0007] Streaming media has quickly become the most popular form of multimedia content on the Internet. Video from news, sports, and entertainment sites are more popular than ever. Media servers are also being used for educational and training purposes by many universities. Further, use of media servers in the enterprise environment is also gaining momentum. Many radio broadcasts are available over the Internet, which make use of streaming audio to enable a much wider audience access to their broadcasts.

[0008] In view of the above, the area of multimedia services in a networked environment is a rapidly expanding field in today's technological world. The delivery of continuous media from a central server complex to a large number of (geographically distributed) clients is a challenging and resource intensive task. Media servers are commonly implemented for providing streaming media to clients. Various streaming media files may be provided concurrently by a media server to various different clients. That is, a plurality of clients may concurrently access streaming media files from the media server. Of course, limits exist as

to how many concurrent streams a media server can support for a given client population. That is, limits exist as to the capacity of a media server for supporting a given "workload" (i.e., a number of concurrent client accesses of streaming media from the media server).

BRIEF SUMMARY OF THE INVENTION

[0009] In certain embodiments of the present invention, a method for managing admission of requests to a streaming media server is provided. The method comprises receiving a new request for a streaming media file to be served by a streaming media server, and performing a resource availability check for the streaming media server to determine whether the streaming media server has sufficient available resources to service the new request. The method further comprises performing quality of service guarantee check for the streaming media server to determine whether acceptance of the new request will violate, at any point in the future, a desired quality of service provided by the streaming media server for any previously accepted requests.

[0010] In certain embodiments, a method for managing admission of requests to a media server comprises receiving a new request for a streaming file to be served by a media server, and determining a cost to the media server for serving the requested streaming file, wherein the cost corresponds to the media server's resources to be consumed in serving the requested streaming file. The method further comprises determining, based at least in part on the cost, whether to admit the new request for service by the media server.

[0011] In certain embodiments, a system is provided that comprises a server having a memory, wherein the server is operable to serve at least one streaming file to clients communicatively coupled thereto. The system further comprises an admission controller operable to receive a new request for a streaming file to be served by the server, determine a cost to the server for serving the requested streaming file, wherein the cost corresponds to the server's resources to be consumed in serving the requested streaming file, and determine, based at least in part on the cost, whether to admit the new request for service by the server.

[0012] In certain embodiments, a method is provided that comprises receiving, at a time T_{cur}, a new request for a streaming file to be served by a media server, and creating a segment-based model of the media server's memory as of time T_{cur}. And, based at least in part

on the segment-based model of the media server's memory, determining whether to accept the received request for service by the media server.

[0013] In certain embodiments, computer-executable software stored to a computer-readable medium is provided. The computer-executable software comprises code for creating a segment-based model of a media server's memory, and code for determining whether to serve a requested streaming file from the media server based at least in part on the segment-based model of the media server's memory.

[0014] In certain embodiments, a cost-aware admission control system is provided that comprises a means for receiving, at a time T_{cur}, a new request for a streaming file to be served by a media server. The cost-aware admission control system further comprises a means for creating a segment-based model of the media server's memory as of time T_{cur}, and a means for determining, based at least in part on the segment-based model of the media server's memory, whether to accept the received request for service by the media server.

BRIEF DESCRIPTION OF THE DRAWINGS

[0015] FIGURE 1 shows an example client-server system in which embodiments of the present invention may be implemented;

[0016] FIGURE 2 shows an example of how file segments are stored to memory when concurrent (or overlapping) accesses are made to different media files, given the real-time nature of such media files;

[0017] FIGURE 3 shows an example in which concurrent (or overlapping) accesses are made to the same file *f*;

[0018] FIGURE 4 shows an example operational flow diagram for determining a data structure that represents the "unique" segments of a file that were accessed during a time interval of interest in accordance with an embodiment of the present invention;

[0019] FIGURE 5 shows an example of requests for file accesses that are made to a media server during the interval of time t_1 through time T, wherein the interval from time T^{mem} through time T can be determined that comprises the segments of accessed files that are currently stored to the media server's memory, which has size $Size^{mem}$;

[0020] FIGURE 6 shows an example of determining a time interval that includes file accesses such that the sum of unique bytes accessed is equal to the size of the media server's memory;

[0021] FIGURE 7 shows an example operational flow diagram for determining the memory state of a media server from which the type of access required for satisfying a received request can be determined in accordance with an embodiment of the present invention;

[0022] FIGURE 8 shows an example flow diagram of determining a media server's memory state and using the memory state to determine a type of access of a newly received request in accordance one embodiment of the present invention;

[0023] FIGURE 9 shows an example operational flow diagram of one embodiment of an admission controller;

[0024] FIGURES 10A-10B show an example operational flow diagram for Quality of Service (QoS) validation that is performed by an admission controller in accordance with one embodiment of the present invention;

[0025] FIGURE 11 shows an example of safely replaceable bytes that may be determined for a media server's memory;

[0026] FIGURES 12A-12B show performance results for two workloads applied in a sensitivity study to an example media server having a cost-aware admission controller of an embodiment of the present invention;

[0027] FIGURE 13 shows an example Utility Data Center arrangement in which resources may be dynamically allocated and in which certain embodiments of the present invention may be implemented; and

[0028] FIGURE 14 shows an example computer system on which embodiments of the present invention may be implemented.

DETAILED DESCRIPTION

[0029] Embodiments of the present invention provide a system and method for managing admission of requests to a streaming media server. For instance, a cost-aware

admission control system for a media server is provided, wherein a cost is determined for a received request and such cost is used to determine whether the media server can support such request. In accordance with an embodiment of the present invention, the current memory state of a media server at any given time may be determined through a memory state model. Such modeling of the media server's memory state provides a close approximation of the real system memory but reflects a higher-level memory abstraction.

[0030] As described further below, certain embodiments of the present invention enable the memory state of a media server to be efficiently determined, which enables such modeling of the memory state to be used, for example, in implementing an admission control system for managing the acceptance of requests to be serviced by the media server. In accordance with embodiments of the present invention, from the modeled memory state, a determination of the streaming files (or file segments) present in the media server's memory at any given time can be intelligently estimated. Accordingly, a determination of the media server's resources that will be consumed in servicing a newly received request may be computed. That is, the memory state may be used to efficiently determine whether the memory resources or the disk resources of a media server will be consumed in servicing a newly received request for a particular streaming file. Thus, a "cost" of the media server's resources required for servicing the received request (e.g., whether memory resources or disk resources will be utilized) may be determined for the request, and such cost may be utilized for determining whether the request should be admitted for service by the media server or whether admitting such request would result in overloading the media server such that the quality of service provided by the media server degrades below a desired level. Thus, modeling of the media server's memory state may be used for implementing an admission control policy for determining whether the media server can service a newly received client request without undesirably degrading the quality of service provided by the media server.

[0031] Certain embodiments of the present invention utilize a segment-based access model for representing unique, most recently accessed segments of a file. Such a segment-based access model may be used for efficiently computing a media server's memory state, as described further below. That is, considering that a streaming file may be concurrently accessed (or have overlapping accesses thereto), various portions (or segments) of the streaming media file may have different time stamps at which they have been most recently accessed by a client. As used herein, a streaming file is intended to encompass any type of file now known or

later discovered that is served in a streaming fashion. Examples of a streaming file include well-known streaming media files, such as streaming audio and/or streaming video files.

[0032] For example, a first client may request, at time t=0 seconds, access to a streaming media file *f* that is 600 seconds in duration, and a second client may request access to such streaming media file *f* 10 seconds later, at time t=10 seconds. Given the real-time nature of streaming media files, at time t=50 seconds, the first client will have accessed 50 seconds of file *f* (i.e., the segment from 0 seconds to 50 seconds of file *f*), and the second client will have accessed 40 seconds of file *f* (i.e., the segment from 0 seconds to 40 seconds of file *f*). However, the second client's access of the segment 0-40 seconds of file *f* is more recent than the first client's access of that segment. That is, the second client's access of the segment 0-40 seconds of file *f* started at time t=10 seconds, whereas the first client's access of that segment started at time t=0 seconds. But, the first client has most recently accessed the segment 40-50 seconds of file *f* (i.e., such access of segment 40-50 seconds of file *f* was started by the first client at time t=40 seconds), and the second client has not yet accessed this segment at all. A segment-based access model of an embodiment of the present invention is utilized for file *f* to represent that segment 0-40 seconds of file *f* was most recently accessed starting at time t=10 seconds and segment 40-50 seconds of file *f* was most recently accessed starting at time t=40 seconds. Again, such a segment-based access model for streaming media files is utilized in certain embodiments of the present invention to enable a memory state of a media server to be efficiently computed.

[0033] Further, by computing the memory state from the segment-based access model in accordance with certain embodiments, a segment-based model of the media server's memory results. That is, a segment-based model of the media server's memory may be determined, as opposed to the real organization (e.g., "page" organization) of the memory. Such a segment-based model of the memory enables efficient determination of the content of such memory over time (e.g., determination of the file segments that are evicted from memory in favor of inserting new file segments into memory, etc.).

[0034] Thus, as described further below, an embodiment of the present invention provides a cost-aware admission control strategy for a media server that efficiently utilizes the available media server resources (e.g., the media server's capacity) to support an optimum number of requests (or an optimum "workload") while ensuring that desired quality of service

("QoS") guarantees are satisfied. Various embodiments of the present invention are now described with reference to the above figures, wherein like reference numerals represent like parts throughout the several views. Turning first to FIGURE 1, an example client-server system 100 is shown in which embodiments of the present invention may be implemented. Client-server system 100 comprises a plurality of clients 104, 105, and 106, and a media server (or complex of media servers) 101. As used herein, a media server (or "streaming media server") is intended to encompass any processor-based device now known or later developed that is capable of serving one or more streaming files to clients thereof. Of course, such a media server (or "streaming media server") may be further operable to also serve other types of files to its clients. Clients 104-106 are each capable of communicatively accessing media server(s) 101 via communication network 103. Communication network 103 is preferably a packet-switched network, and in various implementations may comprise, as examples, the Internet or other Wide Area Network (WAN), an Intranet, Local Area Network (LAN), wireless network, Public (or private) Switched Telephony Network (PSTN), a combination of the above, or any other communications network now known or later developed within the networking arts that permits two or more computing devices to communicate with each other.

[0035] Media server(s) 101 of the illustrated embodiment comprise data storage 102 for storing streaming media files, such as File A, File B, and File C. Data storage 102 may comprise internal or external disk drives, floppy disks, optical disks, Compact Discs (CDs), Digital Versatile Discs (DVD), memory, and/or other data storage devices now known or later developed for storing digital data. For instance, as described further below, data storage 102 typically comprises at least disk resources and memory resources for storing streaming media files (or segments thereof). In operation, a client may request a streaming media file available from media server 101, and such media server 101 may serve the streaming media file as a stream to the requesting client via communication network 103. That is, a client may request a particular content (e.g., audio, video, or other type of content) and media server 101 may serve a streaming media file as a stream to provide the requested content to the requesting client.

[0036] Media server 101 may comprise streaming media file(s) that is/are encoded for transmission at each of a plurality of different bit rates. For example, a streaming media file, File A, may comprise a particular content and it may be encoded for transmission at a plurality of different bit rates, such as 28 Kb/s, 56 Kb/s, and/or various other bit rates. Each resulting version of the file encoded for transmission at a given bit rate may be stored to data storage 102,

e.g., File A_1 encoded for transmission at 28 Kb/s and File A_2 encoded for transmission at 56 Kb/s may each be stored to data storage 102 (note that files A_1 and A_2 comprise substantially the same content but are encoded for transmission at different bit rates, and thus the quality of each file may differ). As used herein, a file encoded for transmission at a particular bit rate may be referred to as a file encoded at the particular bit rate. In common phraseology in the art, a streaming media file is referred to as being "encoded at a particular bit rate", which means the file is encoded for transmission from the server at the particular bit rate. Thus, as used herein, the phrase "encoded at a bit rate" when describing a streaming media file means the streaming media file is encoded for transmission at the bit rate, as is consistent with common phraseology in the streaming media art.

[0037] Media server 101 generally attempts to serve the most appropriate encoded file to a client based at least in part on the client's access speed to the client-server network 103. For example, suppose client 104 has a 28 Kb/s speed connection to communication network 103, client 105 has a 56 Kb/s speed connection to communication network 103, and media server 101 comprises File A_1 encoded at 28 Kb/s and File A_2 encoded at 56 Kb/s stored to data storage 102; when client 104 requests the content of File A, media server 101 typically attempts to serve File A_1 to client 104 (as File A_1 is the highest-quality encoded file supportable by client 104's connection speed), and when client 105 requests the content of File A, media server 101 typically attempts to serve File A_2 to client 105 (as File A_2 is the highest-quality encoded file supportable by client 105's connection speed). However, in the above example, suppose that media server 101 does not have File A encoded at 56 Kb/s but instead comprises File A_1 encoded at 28 Kb/s and File A_2 encoded at 112 Kb/s; then upon client 105 requesting the content of File A, media server 101 typically attempts to serve File A_1 to client 105, as File A_1 is the highest-quality encoded file providing the requested content that is encoded at a suitable bit rate that client 105's connection can support.

[0038] Typically, a streaming media player is executed by the requesting client for playing the received stream. Various types of streaming media technologies exist. Popular streaming media players include those provided by RealNetworks™ (*see* http://www.realnetworks.com), such as its RealPlayer™ and RealOnePlayer™ streaming media players, and that used by Microsoft's Windows® Media Player (*see* http://www.microsoft.com), as examples. For instance, clients 104, 105, and 106 comprise streaming media players 104A, 105A, and 106A, respectively, executing thereon for playing received streams in the example of

FIGURE 1. Typically, each streaming media player has a buffer associated therewith, such as buffers 104B, 105B, and 106B shown in the example of FIGURE 1, for buffering received streamed data to improve the continuity of the playback of such streamed data by the streaming media player.

[0039] As an example of a typical scenario for a user accessing an audio stream via the web, the user will typically use a web browser, such as Netscape's Navigator™, Microsoft's Internet Explorer™, or other web browser now known or later developed, to request access to audio content (e.g., a RealPlayer sound clip) by, for example, clicking a link to such audio content provided on a web page being served by a web server. Assuming, for instance, that the requested audio content is included in a RealPlayer sound file, the web server sends back to the browser a file called a RealPlayer metafile, which is a small text file that has the true location (e.g., the Universal Resource Locator (URL)) of the requested RealPlayer sound file and also has instructions instructing the web browser to launch the RealPlayer sound player. For instance, client 104 may submit request 107 for desired streaming audio content by, for example, clicking on a hyperlink to such streaming audio content. If a suitable streaming media player 104A (e.g., a RealPlayer media player in the above example) is not already executing on client 104, media server 101 may return instructions launching such streaming media player 104A.

[0040] The streaming media player 104A contacts the URL contained in the received metafile. It should be recognized that often the URL contacted for accessing the desired audio stream is not on the web server from which the audio file was requested. Rather, the audio stream is often on a different media server (e.g., a RealPlayer server in the above example) designed to deliver streaming media files. For instance, in FIGURE 1, client 104 may access a web server (not shown) and a user may click on a hyperlink provided on a web page being served by such web server to request desired audio content. The web server may return a metafile to the client indicating the URL of the corresponding sound file and launching streaming media player 104A if it is not already executing. The URL of the sound file may identify media server 101, which is designed for delivering streaming media files, such as those stored to data storage 102, and streaming media player 104A may therefore contact media server 101 (via request 107 in this example).

[0041] Media server 101 (e.g., a RealPlayer server in this example) and streaming media player 104A (e.g., a RealPlayer media player in this example) may interact with each

other so that the server knows at what speed client 104 is connected to the Internet. If the connection is a low-speed connection, a streaming audio file encoded at a lower bit rate is typically sent. This will generally be a file of lesser quality than a file encoded at a higher bit rate and destined for a high-speed connection. If a high-speed connection is used, a larger, higher-quality sound file encoded at a higher bit rate is typically sent, which will generally provide for better sound quality. The requested audio file is then sent in Internet Protocol (IP) packets, typically using either the User Datagram Protocol (UDP) or the Internet's normal Transmission Control Protocol (TCP). UDP does not keep re-sending packets if they are misplaced or other problems occur, as does TCP, which may be preferable for certain streaming media technologies.

[0042] Thus, media server 101 serves the requested audio content to client 104 as stream 108. The packets are sent to buffer 104B on client 104, and once the buffer is filled, the packets begin being sent from the buffer to streaming media player 104A and it begins playing the audio file. As the file is being played remaining portions of the file are still transmitted from media server 101 to client 104 and are buffered in buffer 104B. If packets are not replenished to buffer 104B fast enough, then interruptions in the playback by the streaming media player 104A may occur, thus degrading the quality of the audio stream perceived by the user.

[0043] Steaming video may be communicated from media server 101 to a client in a similar manner as that described above for streaming audio. For instance, client 105 may submit request 109 for desired streaming video content. The requested video file is then sent in IP packets, typically using UDP. Thus, media server 101 serves the requested video file to client 105 as stream 110. The packets are sent to buffer 105B, and once the buffer fills up, the video begins being played by streaming media player 105A. As the file is being played, remaining video packets of the file are still transmitted from media server 101 to client 105 and are buffered in buffer 105B. If packets are not replenished to buffer 105B fast enough, then interruptions in the playback by streaming media player 105A may occur, thus degrading the quality of the video stream perceived by the user. Once all of the video data has been received and played by streaming media player 105A, the video stops. The video file generally does not remain on the client's system, but rather each section of the file is typically discarded after it is played.

[0044] As further shown in FIGURE 1, client 106 may request streaming media content (e.g., a streaming audio and/or video file) from server 101 via request 111, and media

server 101 may provide the requested content as stream 112 to client 106. While three clients are shown in this example, it should be recognized that many more clients may, in certain implementations, be concurrently accessing one or more streaming media files from media server 101. As described above, limits exist as to how many concurrent streams media server 101 can support for a given client population. Thus, it may become desirable to measure the capacity of a media server for supporting workloads applied thereto.

[0045] In order to measure the capacity of a media server for supporting a received client request, for example, it may be desirable to determine the impact that servicing such received client request will have on the media server's resources. That is, it may be desirable to determine whether servicing the received client request will impact the media server's memory resources or disk resources. As described further below, embodiments of the present invention provide a cost-aware admission control system for managing the admission of requests to be serviced by a media server. Certain embodiments utilize a memory model for efficiently modeling the media server's memory, wherein such memory model may be used to determine whether a requested file can be served from the media server's memory or whether such requested file is not present in memory and must be served from the media server's disk. From such determination of whether the requested file can be served from the media server's memory, a resource "cost" may be determined for the request, and such cost may be used in an admission controller to determine whether the request should be accepted for service by the media server.

[0046] Further, while a client requests streaming media content in each of the above examples, it should be recognized that in some implementations a streaming media file may be "pushed" from media server 101 to a client without the client specifically requesting such file. For instance, upon a client visiting a particular web page, the web page may cause a streaming media file (e.g., an advertisement) to be sent to the client. It should be understood that embodiments of the present invention are applicable to any such utilization of streaming media.

[0047] While examples of streaming media technologies, such as typical techniques for accessing RealPlayer streaming media files, are described above, it should be recognized that the present invention is not limited to a specific type of streaming media technology. Further, while examples of streaming audio and streaming video files are described above, the present invention is not limited solely to such types of streaming media, but rather any type of streaming file is intended to be within the scope of the present invention.

[0048] There are different types of media servers available in the art. Examples of known media servers include RealNetwork™ Server 8, Windows™ Media Server, QuickTime™ Streaming Server available from Apple Computer, Inc. In servicing client requests, some media servers, such as early versions of the Windows™ Media Server, completely bypass memory. However, many media servers, such as RealNetwork™ Server 8 use the system's memory in servicing client requests. For instance, many media servers use the native operating system file buffer cache support for buffering recently accessed portions of files being served to clients. Files available in memory can generally be accessed more efficiently than those stored to disk. If the file is not available in memory, then the media server accesses its disk to retrieve the file. As the file (or currently needed portion thereof) is retrieved from disk, it is stored to memory and served to the requesting client. In this manner, many media servers use the system's memory (e.g., native operating system file buffer cache support) for buffering recently accessed portions of files being served to clients to enable later accesses to those portions of files that are stored to memory to be served more efficiently by the media server. Embodiments of the present invention are useful for modeling the memory of such media servers that make use of memory for buffering recently accessed portions of files being served to clients, wherein such modeling may be used for implementing an admission control system for managing the admission of newly received requests for the media server.

[0049] Co-pending U.S. Patent Application Number 10/306,279 filed November 27, 2002 entitled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER," the disclosure of which is hereby incorporated herein by reference, discloses a set of benchmarks for measuring the basic capacities of streaming media systems. The benchmarks allow one to derive the scaling rules of server capacity for delivering media files which are: *i*) encoded at different bit rates, and *ii*) streamed from memory versus disk. As U.S. Patent Application Number 10/306,279 titled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER" further describes, a "cost" function can be derived from the set of basic benchmark measurements. This cost function may provide a single value to reflect the combined resource requirement such as CPU, bandwidth, and memory necessary to support a particular media stream depending on the stream bit rate and type of access (e.g., memory file access or disk file access).

[0050] Traditional media server solutions do not have a built-in admission control policy (for controlling the admission of new client requests to be serviced by the media server)

that can prevent server overload and guarantee a desired quality of service. Instead, traditional media servers continue to admit new client requests upon their receipt by the media server and provide degraded quality of service by sharing the available server resources among the admitted streams (even when the media server's resources are overloaded and unable to support the admitted streams).

[0051] Certain admission control strategies have been proposed in the technical literature that propose using the maximum bandwidth a server is capable of delivering as an indicator for admission control. However, as described in U.S. Patent Application Number 10/306,279 titled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER", the amount of bandwidth a server is capable of delivering is variable and depends on the encoding bit rates of current streams in progress. Another possible approach for admission control strategy is based on a pessimistic model, where a new client request is admitted only if a server has enough resources to deliver this request from the disk. However, media server performance may be 3-5 times higher (depending on disk and file system) when media streams are delivered from memory versus from disk. Thus, while the pessimistic admission control strategy provides hard guarantees against server overload, it may lead to significant over-provisioning and inefficient resource usage. Thus, this strategy may indeed be too "pessimistic" because it has been observed that typical media workload has a high degree of reference locality (i.e., a high percent of requests often are accessing a small subset of media files) and exhibit a strong sharing pattern (i.e., accesses to the same file come in "bursts"). Intuitively, in such a workload, most of the accesses to popular files may be served from memory, rather than from disk.

[0052] Therefore, in order to provide an efficient admission control mechanism for a streaming media server, it is desirable to be capable of estimating the cost of a new request acceptance, and hence to be able to determine which segments of the requested file are stored in memory. More specifically, by determining which segments of a requested file are stored in memory, an estimate of consumption of memory versus disk resources of the media server in serving the requested file can be estimated. Accordingly, certain embodiments of the present invention enable a media server's current memory state to be modeled to enable, for example, a determination of whether a newly received request will initially be serviced by memory. That is, whether the received request will have an immediate impact on memory resources or disk resources of the media server can be efficiently determined from the modeled memory state.

More particularly, the modeled memory state may identify the segments of files stored to the media server's memory (i.e., the content of the memory), and thus from such modeled memory state a determination can be made whether a prefix of a requested file is available in memory. If the prefix of a requested file is in memory then such request, if admitted for service by the media server, will have an immediate impact on the memory resources, whereas if the prefix of the requested file is not in memory then such request, if admitted for service by the media server, will have an immediate impact on disk resources. Thus, as described further below, a memory state model may be used to implement a cost-aware admission control strategy for managing admission of requests to a streaming media server.

[0053] A. Segment-Based Access Model

[0054] To efficiently determine a memory state of a media server, certain embodiments of the present invention utilize a segment-based model (or data structure) of streaming media files, as described further below. That is, an embodiment of the present invention that uses a segment-based model representing streaming media file accesses is described further below, wherein for each client request *r* the following notations are used:

file(r) - the media file requested by *r*;

duration(r) - the duration of file (*r*) in seconds;

bitRate (r) - the encoding bit rate of the media file requested by *r* (in describing this embodiment, it is assumed that files are encoded at constant bit rates);

$t^{start}(r)$ – the time when a stream corresponding to request *r* starts (once *r* is accepted by the media server);

$t^{end}(r)$ – the time when a stream initiated by request *r* is terminated (in describing this embodiment, non-interactive client sessions are assumed, i.e., $duration(r) \leq t^{end}(r) - t^{start}(r)$).

[0055] As an example of the real-time nature of streaming media, let request *r* be a sequential access to streaming media file *f* from the beginning of the file. For simplicity, let it be a disk access. Then, after 10 seconds of access *r*, the content, transferred by a server, corresponds to the initial 10 seconds of the file *f*. The duration of a transferred file prefix defines the number of bytes transferred from disk to memory and further to the client: in this example, it is assumed to be $10\,\text{seconds} \times \text{bitRate(r)}$. Moreover, the real-time nature of file access in streaming media defines the relative time ordering of streamed file segments in

memory. This means that the time elapsed from the beginning of the file (e.g., 0 seconds is used here to denote the file beginning) can be used to describe both: 1) the streamed file segment and 2) the relative timestamps of this file segment in memory.

[0056] To clarify this point further, FIGURE 2 shows an example of how file segments are stored to memory when concurrent (or overlapping) accesses are made to different media files, given the above-described real-time nature of such media files. For the example of FIGURE 2, consider a media server that has a 100 MB memory, and the media server has media files f_1, f_2, and f_3 stored thereto, wherein each of such media files f_1, f_2, and f_3 are 600 seconds (10 minutes) long and encoded at 100 KB/s. Let us consider the following sequence of request arrivals as shown in FIGURE 2:

request r_1 for file f_1 arrives at time $t_1 = 0$ seconds (*sec*);
request r_2 for file f_2 arrives at time $t_2 = 100$ *sec*; and
request r_3 for file f_3 arrives at time $t_3 = 200$ *sec*.

[0057] It should be recognized that the time reference shown in FIGURE 2 is a "global" time reference across all of the received requests. For instance, request r_1 for file f_1 arrives at time $t_1 = 0$ *sec*, and request r_2 for file f_2 arrives 100 seconds later at time $t_2 = 100$ *sec*. As described further below, the file segments may be designated by their "local" time references. For instance, at time t=150 *sec* file f_1, which began being accessed at time $t_1 = 0$ *sec*, is 150 seconds into its stream (which is 600 seconds in duration), and file f_2, which began being accessed at time $t_2 = 100$ *sec*, is 50 seconds into its stream (which is also 600 seconds in duration).

[0058] Let us evaluate the media server's memory state at time point $T_{cur} = 500$ *sec*, shown in FIGURE 2. While the overall number of bytes transferred by the three requests is 120 MB, the memory of this example media server can hold only 100 MB of latest (most recent) portions of transferred files which are represented by the following file segments:

a) a segment of file f_1 between 150 seconds and 500 seconds of its duration. The denotation <150,500>(150) is used herein to describe this segment, where numbers in " <>" describe the beginning and the end of a file segment, and a number in "()" defines a relative timestamp (or "global timestamp") in memory corresponding to the beginning of the file segment. That is, the numbers in " <>" describe the "local" timestamp of a file segment by identifying the beginning and the end times of the file segment, and the number in "()" defines a

"global" timestamp for a time reference that is consistent across all requests;

b) a segment of the file f_2: <50,400>(150), i.e., the segment of file f_2 from 50 seconds to 400 seconds of its duration with the access of such segment starting at the global timestamp of 150 seconds; and

c) a segment of the file f_3: <0,300>(200), i.e., the segment of file f_3 from 0 seconds to 300 seconds of its duration with the access of such segment starting at the global timestamp of 200 seconds.

[0059] This new abstraction provides a close approximation of file segments stored in memory and their relative time ordering (timestamps) in memory. That is, for the example of FIGURE 2, at time $T_{cur} = 500$ *sec*, the following segments are stored to the media server's memory: file f_1 <150,500>(150), file f_2 <50,400>(150), and file f_3 <0,300>(200).

[0060] The above example of FIGURE 2 provides an example of modeling a media server's memory for concurrent accesses (or "overlapping" accesses – i.e., one access beginning before another access ends) to different files f_1, f_2, and f_3. In many instances, the overlapping accesses may be for the same file f. If there are multiple concurrent (or overlapping) accesses to the same file f, then requests with later arrival time might find the corresponding file segments being already in memory. Thus, operations for computing the ***unique segments*** of file f with the most recent timestamps which correspond to a sequence of accesses to f may be utilized in an embodiment of the present invention, as described further below.

[0061] In accordance with an embodiment of the present invention, a file segment transferred by request r^f during time interval $[T, T']$ is defined as follows:

$$segm(r^f, T, T') = <x, y> (\hat{T})$$

where

$$x = \max\{T, t^{start}(r^f)\} - t^{start}(r^f),$$

$$y = \min\{t^{end}(r^f), T'\} - t^{start}(r^f), \text{ and}$$

$$\hat{T} = \max\{T, t^{start}(r^f)\}.$$

[0062] In computation of a current memory state in accordance with an embodiment of the present invention, the "unique" file segments currently present in memory are computed. This means that in a case of multiple requests to the same file, the accesses and the corresponding file segments with the latest access time are identified in a manner that avoids

repetitively counting of the same bytes accessed by different requests at different time points. Thus, file segments of a model (or data structure) are referred to herein as being "unique" because the corresponding portion of the file of each segment is included in only one of the segments of the model. That is, each segment of the access model represents a unique portion of a file, and as described further below, each segment has associated therewith a corresponding global timestamp that identifies the time at which such segment was last accessed.

[0063] As a simple example, suppose a first request r_1 is received at global time t=0 *sec* for a file *f*, which is 100 *sec* in length. Assuming no other requests are made in the interim, at global time 100 *sec* the entire file *f* has been served to the requesting client and saved to the media server's memory. Accordingly, a segment of the memory may be denoted as *f* <0,100> (0), which identifies that segment <0,100> of file *f* (the entire file *f* in this example) is stored to memory having a latest access time of global timestamp t=0 *sec*. Assume now that at global time t=150 *sec* a second request r_2 for file *f* is received by the media server. Because file *f* is available in memory, it can be served to the client from memory. However, the memory segment for file *f* should now be updated to reflect that it has been more recently accessed. Thus, the data structure modeling such segment may be updated to provide: *f* <0,100> (150), which identifies that segment <0,100> of file *f* (the entire file *f* in this example) is stored to memory having a latest access time of global timestamp t=150 *sec*. By updating the global timestamp at which the file segment was most recently accessed, a proper determination may be made as to the file segments (or portions thereof) that may be evicted from the media server's memory in accordance with the memory management scheme implemented for the server, such as a Least Recently Used (LRU) scheme, as described further below.

[0064] To explain this situation in more detail, attention is now directed to FIGURE 3, which graphically depicts an example in which concurrent accesses are made to the same file *f*. In the example of FIGURE 3, let r_1^f, r_2^f, r_3^f be a sequence of requests accessing the same file *f* in the following arrival order: $t^{start}(r_1^f) = 0$ *sec*, $t^{start}(r_2^f) = 10$ *sec*, and $t^{start} = (r_3^f) =$ 20 *sec*. That is, request r_1^f is received at global time T=0 *sec*, request r_2^f is received at global time T=10 *sec*, and request r_3^f is received at global time T=20 *sec*.

[0065] By the time $T' = 50$ *sec*, the first request r_1^f has transferred segment <0,50>(0) of file *f*, and the initial part of this segment <0,40>(0) is again accessed and

transferred at a later time by the second request r_2^f. Thus, segment <40,50>(40) is the only "unique" segment of file f accessed by r_1^f most recently. That is, segment <40,50>(40) is the only segment of file f that has been accessed most recently by request r_1^f because other segments of file f accessed by request r_1^f have been more recently accessed by other requests, such as request r_2^f.

[0066] Similarly, segment <30,40>(40) represents the only unique segment of file f that was accessed most recently by r_2^f. More specifically, by the time $T' = 50\,sec$, request r_2^f has transferred segment <0,40>(10) of file f, and the initial part of this segment <0,30>(10) is again accessed and transferred at a later time by the third request r_3^f. Thus, segment <30,40>(40) is the only unique segment of file f accessed by r_2^f most recently. That is, segment <30,40>(40) is the only segment of file f that has been accessed most recently by request r_2^f because other segments of file f accessed by request r_2^f have been more recently accessed by other requests, such as request r_3^f.

[0067] Finally, the latest request r_3^f is accountable for the most recent access to the initial segment <0,30>(20) of file f. Thus overall, the unique segments of file f with the most recent timestamps in the global time reference $[0, 50]sec$ interval are the following:

segm(f, 0, 50)= {<0,30>(20), <30,40>(40), <40,50>(40)}.

[0068] In the above denotation, *segm(f,* 0, 50) identifies that the segment under evaluation is the segment for file f as accessed during the global time reference of 0 *sec* to 50 *sec*. In the example of FIGURE 3, the resulting segments of file f for this evaluation are {<0,30>(20), <30,40>(40), <40,50>(40)}, wherein the segment <0,30>(20) corresponds to the portion of file f most recently accessed by request r_3^f, the segment <30,40>(40) corresponds to portion of file f most recently accessed by request r_2^f, and the segment <40,50>(40) corresponds to portion of file f most recently accessed by request r_1^f.

[0069] To determine the unique, most recent segments of file f accessed by subsequent requests $r_{i_1}^f$ and $r_{i_2}^f$ in $[T,T']$ time interval, a new operation called "segments

subtraction" and denoted as " \ " is introduced herein. Let $r_{i_1}^f$ and $r_{i_2}^f$ be two subsequent requests accessing the same file *f* such that $t^{start}(r_{i_1}^f) \leq t^{start}(r_{i_2}^f)$, i.e. $r_{i_2}^f$ is more recent access than $r_{i_1}^f$. Let $segm_{i_1} = segm(r_{i_1}^f, T, T') = <x_{i_1}, y_{i_1}> (T_{i_1})$ and $segm_{i_2} = segm(r_{i_2}^f, T, T') = <x_{i_2}, y_{i_2}> (T_{i_2})$. Then

$$segm_{i_1} \setminus segm_{i_2} = \begin{cases} <x_{i_1}, y_{i_1}> (T_{i_1}) & \text{if } y_{i_2} \leq x_{i_1} \\ <y_{i_2}, y_{i_1}> (T'_{i_1}) & \text{otherwise} \end{cases} \qquad (1)$$

where $T'_{i_1} = T_{i_1} + (y_{i_2} - x_{i_2})$.

[0070] Intuitively, operation $(segm_{i_1} \setminus segm_{i_2})$ tries to define a part of older segment $segm_{i_1}$, which does not coincide with any part of more recent segment $segm_{i_2}$. Accordingly, this operation results in determination of "unique" file segments that have been accessed.

[0071] Now, let $r_1^f, r_2^f, \ldots, r_n^f$ be a sequence of requests accessing the same file *f* during [T,T'] interval, where $t^{start}(r_1^f) \leq t^{start}(r_2^f) \leq \ldots \leq t^{start}(r_n^f)$, i.e. r_1^f is the oldest access and r_n^f is the most recent access to file *f* in [T, T'] interval. It is desirable to compute the unique segments of file *f* with the most recent timestamps which correspond to requests $r_1^f, r_2^f, \ldots r_n^f$ during time interval $[T, T']$. The general formula to compute such file segments is defined in the following way:

$$segm(f, T, T') = \bigcup_{i=1}^{n} \left(segm(r_i^f, T, T') \setminus segm(r_{i+1}^f, T, T') \right) \qquad (2)$$

where $segm(r_{n+1}^f, T, T') = <0,0> (0)$, i.e. a "zero" size segment. If r^f is the only request accessing file f during $[T, T']$ interval then $segm(f, T, T') = segm(r^f, T, T')$.

[0072] As a further illustrative example, let $segm(f,T,T') = \{<x_1,y_1>(T_1), \ldots, <x_n,y_n>(T_n)\}$. Note that the set of segments *segm(f,T,T')* can be ordered in two different ways: 1) according to file *f*'s structure and/or 2) according to their timestamps, as described further hereafter. The first ordering technique is referred to as "file structure ordering of the segments," in which the segments are ordered according to file *f*'s structure. In an embodiment of the present invention, the segments are ordered according to file *f*'s structure if for any two consecutive segments $<x_i,y_i>(T_i)$ and $<x_{i+1},y_{i+1}>(T_{i+1})$ the following condition holds: $y_i \leq x_{i+1}$, $1 \leq i \leq n-1$. This representation conveniently reflects which

segments of file f were accessed during the time interval $[T,T']$. To distinguish this file structure ordering, the denotation $segm_{order}^{file}(f,T,T')$ is used herein.

[0073] The second ordering technique is referred to as "timestamp ordering of segments" (or "memory ordering of segments"), in which the segments are ordered according to their global timestamps. In an embodiment of the present invention, the segments are ordered according to their global timestamps if for any two consecutive segments $<x_i,y_i>(T_i)$ and $<x_{i+1},y_{i+1}>(T_{i+1})$ the following condition holds: $T_i \le T_{i+1}$, $1 \le i \le n-1$. This representation conveniently reflects the time ordering of accessed file segments during the time interval $[T,T']$. To distinguish time ordering, the denotation $segm_{order}^{time}(f,T,T')$ is used herein.

[0074] The above-described computations may be used to create a *segment-based access model* for the streaming media files, which can be used to efficiently determine the file segments stored in memory (i.e., to compute a current memory state of the media server). More specifically, the above-described computations can be used to create data structures representing segments of a file accessed by one or more requests during a given time interval of interest, and such segment-based model of the file accesses can be used to efficiently determine the current memory state of a media server.

[0075] For instance, FIGURE 4 shows an example operational flow diagram for determining a data structure that represents the "unique" segments of a file that were accessed during a time interval of interest in accordance with an embodiment of the present invention. Operation starts in operational block 401, and in operational block 402 a time interval of interest is determined. For instance, a user or an application, such as an admission control application, may specify a time interval of interest. In operational block 403, the requests accessing a file f during the time interval of interest are identified. That is, the requests for file f received by the media server during the time interval of interest are identified.

[0076] In block 404, it is determined whether at least one request was made for file f during the time interval of interest. If not, then it is determined that there were no requests made for file f during the time interval of interest in block 405 and operation ends at block 410.

[0077] If it is determined in block 404 that at least one request was made for file f during the time interval of interest, operation advances to block 406 whereat it is determined

whether multiple requests were made for file f during the time interval of interest. If not, then the segment of file f accessed by the single request during the time interval of interest is determined in block 407 - recall from above that if r^f is the only request accessing file f during $[T,T']$ interval then $segm(f,T,T') = segm(r^f,T,T')$, and operation advances to block 409, which is described further below.

[0078] If it is determined in block 406 that multiple requests accessed file f during the time interval of interest, operation advances to block 408 whereat the unique, most recent segments of file f accessed by the requests are determined. For instance, in the example of FIGURE 3, a time interval from 0 seconds to 50 seconds is of interest (i.e., [0,50]), and requests r_1^f, r_2^f, r_3^f are identified as being made for file f during such time interval. As described above, the unique, most recent segments of file f accessed by requests r_1^f, r_2^f, r_3^f in the example of FIGURE 3 are {<40,50>(40), <30,40>(40), <0,30>(20)}. Again, these segments are referred to herein as being "unique" because the corresponding portion of the file of each segment is included in only one of the segments of the access model (or data structure). That is, each segment represents a unique portion of file f, and as described above, each segment has associated therewith a corresponding global timestamp that identifies the time at which such segment was last accessed.

[0079] As shown in the example of block 408, in certain embodiments blocks 408A and 408B may be performed for determining the unique, most recently accessed segments of file f. In block 408A, the segment of file f accessed during the time interval of interest by each request is determined. For instance, in the example of FIGURE 3, a segment <0,50>(0) is accessed by request r_1^f, a segment <0,40>(10) is accessed by request r_2^f, and segment <0,30>(20) is accessed by request r_3^f during the time interval [0,50].

[0080] In block 408B, segment subtraction is performed to determine the unique, most recently accessed segments of file f. For instance, in the example of FIGURE 3, segment subtraction is performed, and it is determined that the unique, most recent segments of file f accessed by requests r_1^f, r_2^f, r_3^f are {<40,50>(40), <30,40>(40), <0,30>(20)}. As described above, the unique segments of file f with the most recent timestamps corresponding to requests $r_1^f, r_2^f, \ldots r_n^f$ during a time interval $[T,T']$ of interest may be computed using the formula:

$$segm(f,T,T') = \bigcup_{i=1}^{n}\left(segm\left(r_i^f,T,T'\right) \setminus segm\left(r_{i+1}^f,T,T'\right)\right) \qquad (2)$$

where $segm\left(r_{n+1}^f,T,T'\right) = <0,0>(0)$, i.e. a "zero" size segment.

[0081] In operational block 409, a data structure representing the segments of file f accessed during the time interval of interest (e.g., $segm(f,T,T')$) is created. For instance, continuing with the example of FIGURE 3, a data structure $segm(f,0,50)$={<40,50>(40), <30,40>(40), <0,30>(20)} is created. As described above, in certain embodiments, the segments of $segm(f,T,T')$ may be ordered in two different ways: 1) according to file f's structure and/or 2) according to the segments' respective timestamps. Operation may then end in block 410.

[0082] B. Computing Memory State for a Media Server

[0083] As described with FIGURE 4 above, data structures may be created that represent the unique, most recently accessed segments of a file for accesses occurring during a time interval of interest. In accordance with an embodiment of the present invention, such data structures may be used to compute (or model) the current memory state of a media server. As an example, suppose a media server has files f_1, f_2, f_3, f_4, and f_5 stored thereto, which are each encoded at the same bit rate (e.g., 56 Kb/s). Further suppose that the files have the following durations: f_1 = 45 *sec* duration, f_2 = 200 *sec* duration, f_3 = 360 *sec* duration, f_4 = 240 *sec* duration, and f_5 = 100 *sec* duration. The above-described modeling technique may be performed to create a segment-based access model for each file. For instance, a segment-based access model may be created for each of files f_1 - f_5 based on accesses to those files during the time interval [0,5000] (i.e., from global timestamp 0 *sec* through global timestamp 5000 *sec*). The resulting segment-based access models that may be created for each of the files in this example are as follows:

$segm(f_1, 0, 5000)$ = {<0,40>(4960), <40,45>(4995)};

$segm(f_2, 0, 5000)$ = {<0,200>(270)};

$segm(f_3, 0, 5000)$ = {<0,360>(4500)};

$segm(f_4, 0, 5000)$ = {<0,240>(3560)}; and

$segm(f_5, 0, 5000)$ = {<0,100>(1025)}

[0084] Thus, a segment-based access model (or data structure) is constructed for each of the files f_1 - f_5 that identifies unique, most recently accessed segments of each file during the time interval [0,5000]. As can be seen in the above segment-based access model for file f_1,

its segment <0,40> was last accessed starting at time 4960 *sec* and its segment <40,45> was last accessed starting at time 4995 *sec*. From the segment-based access model of file f_2, it can be seen that its entire file (i.e., segment <0,200>) was last accessed starting at time 270 *sec*. From the segment-based access model of file f_3, it can be seen that its entire file (i.e., segment <0,360> was last accessed starting at time 4500 *sec*. From the segment-based access model of file f_4, it can be seen that its entire file (i.e., segment <0,240>) was last accessed starting at time 3560 *sec*. Finally, from the segment-based memory model of file f_5, it can be seen that its entire file (i.e., segment <0,100>) was last accessed starting at time 1025 *sec*.

[0085] Suppose now that the media server's memory has a size that is capable of storing up to 400 seconds of the streaming media files (i.e., $\text{memory size} = 400 \text{ seconds} \times 56\text{Kb/s} = 22400 \text{ Kb}$ in this example), the above segment-based access models may be useful in determining the data stored to the media server's memory at time t=5000 *sec*. For simplicity, this example assumes that all of the files have the same encoding bit rate. Of course, in many streaming media applications the streaming media files may have different encoding bit rates, and thus the amount memory consumed by an access to any such file may be determined as a function of the duration of the file access and the file's encoding bit rate (e.g., $\text{memory consumed for an access} = \text{file access duration} \times \text{file encoding bit rate}$). In this example, all of files f_1 - f_5 cannot be stored in full to the media server's memory because the sum of the bytes accessed for such files f_1 - f_5 (i.e., $945 \text{ sec} \times 56 \text{ Kb/s encoding rate} = 52920 \text{ Kb}$ in this example) exceeds the total bytes capable of being stored to the media server's memory (i.e., 22400 Kb in this example). Thus, it may be desirable to know the content of the media server's memory at time t=5000 (i.e., it may be desirable to know the portions of files f_1-f_5 that are stored to the media server's memory at time t=5000).

[0086] With knowledge regarding the media server's memory management scheme, the above segment-based access model of files f_1-f_5 may be used to determine the state of the media server's memory at time t=5000. For instance, typically a media server's memory is managed in accordance with an LRU scheme, wherein the most recently accessed segments of files are stored to memory and the oldest (or least recently) accessed segments of files are evicted when needed to make room in memory for storing more recently accessed files. Assuming such an LRU scheme is followed for the above example, the state of the media server's memory at time t=5000 can be determined using the segment-based access models of files f_1-f_5. For

instance, from the above segment-based access models of files f_1-f_5, it can be seen that portions <40,45> and <0,40> of file f_1 were most recently accessed, i.e., at times 4995 *sec* and 4960 *sec* respectively. Thus, file f_1 having a duration of 45 *sec* is included in the media server's memory at time 5000 *sec* in this example.

[0087] The next most recent file segment accessed in the above example was the entire file f_3 (i.e., segment <0,360>) which was accessed at time 4500 *sec*. File f_3 has a total duration of 360 *sec*, while the duration of file f_1 is 45 *sec*. Thus, the sum duration of files f_3 and f_1 is 405 *sec*. Because each of the files are assumed in this example to have an encoding bit rate of 56Kb/s, the entire 405 *sec* duration of files f_1 and f_3 exceeds the media server's memory. Thus, the entire 300 *sec* duration of file segment <0,300> of file f_3 cannot be stored to the media server's memory, but rather only the most recent 295 *sec* of such segment is stored to the media server's memory. The remainder of such segment, as well as the least recently accessed segments of files f_2, f_4, and f_5, would have been evicted from memory in order to store the more recent accesses in accordance with the LRU management scheme. Thus, the portion of segment <0,300> of file f_3 that would remain in memory is <5,300>(4505) – it should be noted that the timestamp for the access of this segment is 4505 *sec*. Thus, the resulting contents of the media server's memory in the above example would be as follows:

memory state = {f_1<40,45>(4995), f_1<0,40>(4960), f_3<5,30>(4505)}.

[0088] Computing the current memory state may be described as follows: let $Size^{mem}$ be the size of a media server's memory in bytes and let $r_1(t_1), r_2(t_2), \ldots, r_k(t_k)$ be a recorded sequence of requests to the media server; given the current time T, some past time T^{mem} is computed such that the sum of the bytes stored in memory between T^{mem} and T is equal to $Size^{mem}$. In this manner, the files' segments streamed by the media server between times T^{mem} and T will be in the media server's memory. By modeling the current state of the media server's memory, an intelligent determination of the server's resources that will be utilized to service a newly received client request can be made. That is, an intelligent determination can be made as to whether a newly received client request can be serviced from the media server's memory or whether the received client request will require access to the media server's disk. An example of computing a current memory state of a media server in accordance with an embodiment of the present invention is described further below in conjunction with FIGURES 5-8.

[0089] Turning to FIGURE 5, an example of requests for file accesses that are made to a media server during the interval of time t_1 through time T is shown, wherein the interval from time T^{mem} through time T can be determined that comprises the segments of accessed files that are currently stored to the media server's memory, which has size $Size^{mem}$. More specifically, accesses $r_1, r_2, \ldots, r_{k-1}, r_k$ are made during the time interval from time t_1 through the current time T. As shown in the example of FIGURE 5, the total size of the segments accessed is greater than the total size $Size^{mem}$ of the media server's memory. Thus, depending on the type of memory management scheme implemented for the memory, some of the accessed segments are evicted from the memory. That is, not all of the accessed segments can be stored to memory because the segments' total size is greater than the size $Size^{mem}$ of memory. Typically, a LRU scheme is implemented for a media server, wherein the most recently accessed segments are stored to memory and the oldest (or least recently accessed) segments are evicted when necessary to make room for more recently accessed segments to be stored in memory. To determine the current contents of memory, the time interval from time T^{mem} to the current time T in which unique file segments that have a size totaling size $Size^{mem}$ is determined.

[0090] Because, as described above, the function $segm(f, T^{mem}, T)$ represents the unique segments of file f accessed in $[T^{mem}, T]$ interval, the total amount of unique bytes of file f accessed and stored in memory between $[T^{mem}, T]$ interval can be computed, and such total amount is denoted herein as $UniqueBytes(f, T^{mem}, T)$.

[0091] To determine a media server's memory state in an efficient way, an embodiment of the present invention utilizes an induction-based algorithm for computing the memory state at any given time. As an example, let T_{cur} be a current time corresponding to a new request r^f_{new}, arrival, wherein an admission controller may be implemented to decide whether to accept or reject request r^f_{new} for processing. Further, let T_{prev} denote the time of the previous arrival event, and let T^{mem}_{prev} be a previously computed time such that the sum of bytes stored in memory between T^{mem}_{prev} and T_{prev} is equal to $Size^{mem}$, as shown in FIGURE 6. It should be understood that while the duration from T^{mem}_{prev} to T_{prev} is shown in FIGURE 6 as being the same as the duration from T^{mem}_{cur} to T_{cur}, this will not always be the case. Indeed, often the

duration of interval $[T_{prev}^{mem}, T_{prev}]$ will be different than the interval $[T_{cur}^{mem}, T_{cur}]$. For instance, if the files accessed during each time interval are encoded at different rates, the durations from T_{prev}^{mem} to T_{prev} and from T_{cur}^{mem} to T_{cur} in which the amount of unique bytes accessed in each time segment is equal to $Size^{mem}$ will not be the same. It should be recalled that to determine the amount of memory consumed by requests (i.e., to identify whether the requests fill the $Size^{mem}$) each file request may be computed as a function of the file's encoding rate and duration of its access (e.g., encoding rate of file $\times$ duration of access of the file).

[0092] Upon receipt of r_{new}^{f} at time point T_{cur}, the following are computed:

a) an updated time T_{cur}^{mem} such that the sum of bytes stored in memory between T_{cur}^{mem} and T_{cur} is equal to $Size^{mem}$; and

b) an updated information on file segments stored in memory in order to determine the access type of new request r_{new}^{f} (e.g., whether the requested file will begin being served from memory or from disk if request r_{new}^{f} is admitted to the media server).

[0093] For illustration, let $Files(T_{prev}) = \{f_{i_1}, \ldots, f_{i_k}\}$ be a set of files accessed during $[T_{prev}^{mem}, T_{prev}]$ interval, and let $Reqs(T_{prev})$ be a sequence of requests $r_1, r_2, \ldots, r_n$ accessing files $Files\ (T_{prev})$ during $[T_{prev}^{mem}, T_{prev}]$ interval. To compute the current memory state in time T_{cur} in accordance with an embodiment of the present invention, the following operations are performed:

1) Using Formula (2) from above, for each file $f \in Files(T_{prev})$, all of the corresponding requests $Regs^{f}(T_{prev}) = \{r_1^f, r_2^f, \ldots r_n^f \mid r_1^f \in \mathrm{Re}qs(T_{prev})\}$ are considered, and all of the unique segments of f which were accessed by requests from $Regs^{f}(T_{prev})$ in $[T_{prev}^{mem}, T_{cur}]$ interval are determined. Using *file structure ordering* of segments, the information about the files and their unique segments are stored in a data structure called $FileTable(T_{prev}^{mem}, T_{cur})$:

$$f_{i_1} : segm_{order}^{file}(f_{i_1}, T_{prev}^{mem}, T_{cur})$$

$$\ldots$$

$$f_{i_k} : segm_{order}^{file}(f_{i_k}, T_{prev}^{mem}, T_{cur}).$$

2) Using *timestamp ordering* of segments, the information about the files and their

unique segments are also stored in a data structure called $TimeFileTable(T_{prev}^{mem}, T_{cur})$:

$$f_{i_1} : segm_{order}^{time}(f_{i_1}, T_{prev}^{mem}, T_{cur})$$

$$\ldots$$

$$f_{i_k} : segm_{order}^{time}(f_{i_k}, T_{prev}^{mem}, T_{cur}).$$

3) At this point, all of the necessary information is available to compute an updated time T_{prev}^{mem} such that the sum of unique bytes transferred during the period $[T_{prev}^{mem}, T_{cur}]$ is equal to $Size^{mem}$.

[0094] Using $FileTable(T_{prev}^{mem}, T_{cur})$, the total amount of unique bytes accessed during this time interval is computed as:

$$UniqueBytes(T_{prev}^{mem}, T_{cur}) = \sum_{f \in F_{prev}} UniqueBytes(f, T_{prev}^{mem}, T_{cur}).$$

The difference $UniqueBytes(T_{prev}^{mem}, T_{cur}) - Size^{mem}$ defines by "how much" T_{prev}^{mem} should be advanced to a new time point T_{cur}^{mem}. Using $TimeFileTable(T_{prev}^{mem}, T_{cur})$, which provides the information about file segments according to their time ordering, T_{cur}^{mem} can be determined.

[0095] After that, the corresponding data structures $TimeFileTable$ and $FileTable$ are updated for time interval $[T_{cur}^{mem}, T_{cur}]$ to contain only file segments starting at time T_{cur}^{mem} and later. Data structures $Files(T_{cur})$ and $Reqs(T_{cur})$ are also updated as follows: a) $Files(T_{cur})$ has only files with segments accessed at time T_{cur}^{mem} and later, and b) $Reqs(T_{cur})$ has only requests which contribute to a set of unique file segments in $[T_{cur}^{mem}, T_{cur}]$ interval.

[0096] From the determined memory state, a determination can be made as to whether a file f (or its initial prefix) requested by r_{new}^{f} is residing in memory or not, and thus whether the request r_{new}^{f} will have a type of access to memory or disk correspondingly.

[0097] FIGURE 7 shows an example operational flow diagram for determining the memory state of a media server from which the type of access required for servicing a received request can be determined in accordance with an embodiment of the present invention. In operational block 701, a time interval is determined that includes file accesses that include an amount of unique bytes equaling the size of memory. For instance, as described above with

FIGURE 6 a time interval $\left[T_{cur}^{mem}, T_{cur}\right]$ is determined. In operational block 702, the unique segments of files accessed during the determined time interval (e.g., during $\left[T_{cur}^{mem}, T_{cur}\right]$) are determined. Thus, once this determination is made in block 702, the file segments that are in memory as of the end of the determined time interval (e.g., as of T_{cur}) are known. That is, the memory state of the media server as of the end of the determined time interval is known. In operational block 703, the determined unique segments of block 702 may be used to determine a type of file access that will be initiated for a client request if such request is initiated at the end of the determined time interval (e.g., at T_{cur}). More specifically, the determined memory state (or unique file segments determined as residing in memory) may be evaluated to determine whether, if a requested file access is admitted to the media server, such requested file will result in the file beginning to be served from the media server's memory or whether it will result in an access of the media server's disk being initiated.

[0098] Turning now to FIGURE 8, an example flow diagram of one embodiment of the present invention is shown in greater detail. As shown, at block 801, a new request r_{new}^{f} requesting access to a streaming media file *f* is received by a media server (or admission controller) at time point T_{cur}. In block 802, a time interval $\left[T_{cur}^{mem}, T_{cur}\right]$ is determined that includes file accesses with an amount of unique bytes equaling the size $Size^{mem}$ of the media server's memory.

[0099] As further shown in FIGURE 8, the time interval $\left[T_{cur}^{mem}, T_{cur}\right]$ may be determined in block 802 through operations 802A-802E. In block 802A, the unique segments and corresponding access timestamps for files accessed during [T_{prev}^{mem} , T_{cur}] interval are identified. For illustration, let $Files(T_{prev}) = \{f_{i_1}, \ldots, f_{i_k}\}$ be a set of files accessed during [T_{prev}^{mem} , T_{prev}] interval, and let $Reqs(T_{prev})$ be a sequence of requests $r_1, r_2, \ldots, r_n$ accessing files *Files* (T_{prev}) during [T_{prev}^{mem} , T_{prev}] interval. As described above with FIGURE 6, such accesses during the earlier [T_{prev}^{mem} , T_{prev}] interval may have previously been determined as including unique file segments having a total of bytes equal to the size $Size^{mem}$ of the media server's memory. Thus, in block 802A, formula (2) from above may be used for each file $f \in Files(T_{prev})$ to consider all of the corresponding requests $Regs^{f}(T_{prev}) = \{r_1^f, r_2^f, \ldots r_n^f \mid r_1^f \in \mathrm{Re}qs(T_{prev})\}$, and

all of the unique segments of f which were accessed by requests from $Reqs^f(T_{prev})$ in $\left[T_{prev}^{mem}, T_{cur}\right]$ interval are determined.

[0100] It should be recognized that if accesses during the $[T_{prev}^{mem}, T_{prev}]$ interval have previously been determined as including unique file segments having a total of bytes equal to the size $Size^{mem}$ of the media server's memory, accesses that are older than T_{prev}^{mem} need not be considered in identifying the accesses that correspond to the content of the media server's memory at time T_{cur}. However, accesses from time T_{prev} to time T_{cur} are to be taken into consideration. Thus, all of the unique segments of file f which were accessed by requests from $Reqs^f(T_{prev})$ in $\left[T_{prev}^{mem}, T_{cur}\right]$ interval are identified in block 802A.

[0101] In block 802B, the determined unique segments of interval $\left[T_{prev}^{mem}, T_{cur}\right]$ are stored to a data structure called $FileTable\left(T_{prev}^{mem}, T_{cur}\right)$ (as described above) using file structure ordering of the segments. In block 802C, the determined unique segments of interval $\left[T_{prev}^{mem}, T_{cur}\right]$ are also stored to a data structure called $TimeFileTable\left(T_{prev}^{mem}, T_{cur}\right)$ (as described above) using timestamp ordering of the segments.

[0102] In operational block 802D, the $FileTable\left(T_{prev}^{mem}, T_{cur}\right)$ data structure is used to compute the total amount of unique bytes accessed during the $\left[T_{prev}^{mem}, T_{cur}\right]$ time interval as:

$$UniqueBytes\left(T_{prev}^{mem}, T_{cur}\right) = \sum_{f \in F_{prev}} UniqueBytes\left(f, T_{prev}^{mem}, T_{cur}\right).$$

Thus, $UniqueBytes\left(T_{prev}^{mem}, T_{cur}\right)$ provides the total number of unique bytes of file segments accessed during the time interval $\left[T_{prev}^{mem}, T_{cur}\right]$. Of course, such number of unique bytes may exceed the amount that the media server's memory is capable of storing. Thus, in block 802E, the difference $UniqueBytes\left(T_{prev}^{mem}, T_{cur}\right) - Size^{mem}$ is computed to determine the amount that T_{prev}^{mem} should be advanced to a new time point T_{cur}^{mem} such that the amount of unique bytes accessed between the determined T_{cur}^{mem} and T_{cur} equals $Size^{mem}$.

[0103] Operation then advances to block 803 whereat the unique file segments and corresponding access timestamps for file accesses in the determined time interval $\left[T_{cur}^{mem}, T_{cur}\right]$ are

determined, which provides the current memory state of the media server's memory (as of time T_{cur}). As shown in block 803A, the operation of block 803 may comprise updating data structures *TimeFileTable* and *FileTable* for time interval $[T_{cur}^{mem}, T_{cur}]$ to contain only file segments starting at time T_{cur}^{mem} and later.

[0104] In operational block 804, the determined current memory state may be used to determine the type of access for the newly received request r_{new}^{f}. That is, from the determined memory state, a determination can be made as to whether file f (or its initial prefix) requested by r_{new}^{f} is residing in memory, and thus whether the request r_{new}^{f} will have a type of access to memory or disk correspondingly. For instance, block 804 may comprise the operation of block 804A in which it is determined whether at least a prefix of the requested file f is included in the unique file segments of the current memory state, wherein if at least a prefix of the file f is included therein then r_{new}^{f} is determined to be a memory access type and otherwise r_{new}^{f} is determined to be a disk access type.

[0105] Operation may then advance to block 805 where it is determined whether a new request is received. As shown, operation may wait at block 805 for a time at which a new request is received. Upon a new request being received, operation advances from block 805 to block 806 whereat T_{cur} is set to the current time at which the new request is received, and T_{prev}^{mem} is set to T_{cur}^{mem} (i.e., the previously determined T_{cur}^{mem} now becomes T_{prev}^{mem}). Operation then returns to block 802 to compute the new $[T_{cur}^{mem}, T_{cur}]$ time interval that includes file accesses with the amount of unique bytes equaling the size $Size^{mem}$, determine the new memory state in block 803, and determine the type of access for the newly received request in block 804.

[0106] C. Cost-aware Admission Control for a Media Server

[0107] Embodiments of the present invention provide a cost-aware admission control strategy for streaming media servers that efficiently utilizes the available media server resources (capacity), while ensuring that desired QoS guarantees are maintained. In order to assign a *cost* to a newly received request, a determination is made as to whether the new request will be streaming data from memory (i.e., will have a cost of memory access) or will be accessing data from disk (i.e., will have a cost of disk access). It should be understood that

assigning a "memory access" cost to a newly received request in certain embodiments does not assume or require that the whole file requested resides in memory. For instance, as described above with the example of FIGURE 3, when there is a sequence of accesses to the same file, the first access may read the file from disk (to memory) while all of the subsequent requests access the corresponding file prefix from memory.

[0108] A satisfactory technique for measuring the capacity of a media server under realistic workloads is not available in the prior art. A standard commercial stress test used by most media server vendors measures a maximum number of concurrent streams deliverable by the server when all of the clients are accessing the same file encoded at a certain fixed bit rate, e.g. 500 Kb/s. That is, in standard commercial stress tests, vendors use a particular streaming media file that is encoded for transmission at a particular bit rate for measuring the maximum number of concurrent streams that clients can retrieve of this file.

[0109] The standard commercial stress test approach is unsatisfactory for several reasons. First, media files are often encoded for transmission at different bit rates. For instance, clients in a given population have different speed connections to the Internet (or other client-server network). For example, in a realistic population of clients, different clients typically comprise various different speed connections, such as dial-up modem connections (e.g., using a 28 or 56 kilobits analog modem), Integrated Services Digital Network (ISDN) connections, cable modem connections, Digital Subscriber Line (DSL) connections, and even higher-bandwidth connections, as examples. Accordingly, different clients may have different speed connections to the Internet varying, for example, from 28 kilobits (Kb) per second to 500 Kb (or more) per second, thus resulting in requirements for different bit rate encodings at the streaming media files being accessed by the different clients. That is, a media server may comprise streaming media files encoded for transmission at various different bit rates (e.g., 28 Kb/s, 56 Kb/s, etc.), and may attempt to serve the most appropriate encoded file to a client based at least in part on the client's connection speed to the Internet.

[0110] Additionally, clients typically may access different media files from the server. That is, a media server commonly provides a plurality of different media files, various ones of which may be accessed concurrently by different clients. When concurrent accesses of a single file is measured, it leads to measurement of the server's capacity for serving a streaming media file from memory, which is often not actually the case. Accordingly, the standard

commercial stress tests used by most media server vendors are unsatisfactory for obtaining an accurate measurement of the capacity of a media server for supporting a realistic workload in serving streaming media files to clients.

[0111] As mentioned above, a technique for measuring server capacity using a cost function is disclosed in co-pending U.S. Patent Application Number 10/306,279 filed November 27, 2002 entitled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER," the disclosure of which has been incorporated herein by reference. Also, a technique for measuring server capacity using a cost function is described by L. Cherkasova and L. Staley in "Building a Performance Model of Streaming Media Applications in Utility Data Center Environment", *Proc. of ACM/IEEE Conference on Cluster Computing and the Grid (CCGrid)*, May, 2003 (hereinafter referred to as "the L. Cherkasova Paper"), the disclosure of which is hereby incorporated herein by reference. The above references introduce a basic benchmark that can be used to establish the scaling rules for server capacity when multiple media streams are encoded at different bit rates. For instance, a basic benchmark may be executed for each of various different encoding bit rates for files stored at a media server. An objective of the basic benchmark according to one embodiment is to define how many concurrent streams of the same bit rate can be supported by the media server without degrading the quality of any streams. In accordance with one embodiment, the basic benchmark comprises two types of benchmarks:

1) *Single File Benchmark* measuring a media server capacity when all the clients in the test are accessing the same file, and

2) *Unique Files Benchmark* measuring a media server capacity when each client in the test is accessing a different file.

[0112] Thus, a *Single File Benchmark* (SFB) may be executed for each of various different encoding bit rates for files stored at a media server under evaluation. The SFB measures the media server capacity when all of the clients in the test are accessing the same file. That is, the result of the SFB for a particular encoding bit rate defines the maximum number of concurrent streams of a single file encoded at that particular bit rate that the media server can support. Example techniques for executing SFBs for a media server are described further in co-pending U.S. Patent Application Number 10/306,279 entitled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER."

[0113] Similarly, a *Unique File Benchmark* (UFB) may be executed for each of various different encoding bit rates for files stored at a media server under evaluation. The UFB measures the media server capacity when all of the clients in the test are accessing different files. That is, the result of a UFB for a particular encoding bit rate defines the maximum number of concurrent streams, each of different files that are encoded at the particular bit rate, that the media server can support. Example techniques for executing UFBs for a media server are described further in co-pending U.S. Patent Application Number 10/306,279 entitled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER."

[0114] When all of a media server's clients are accessing a single file (as measured by the SFB), the media server is capable of serving the currently streamed bytes of the file from memory. However, when all of its clients are accessing a different file (as measured by the UFB), the media server serves each file from disk. Thus, the SFB is essentially a best-case scenario benchmark, whereas the UFB is essentially a worst-case scenario benchmark.

[0115] Using an experimental testbed with standard components available in a Utility Data Center environment and proposed set of basic benchmarks, the capacity and scaling rules of media server running RealServer 8.0 from RealNetworks was measured in the L. Cherkasova Paper. The measurement results reported in the L. Cherkasova Paper show that these scaling rules are non-trivial. For example, the difference between the highest and lowest bit rate of media streams used in those experiments was 18 times. However, the difference in maximum number of concurrent streams a server is capable of supporting for corresponding bit rates is only around 9 times for a *Single File Benchmark,* and 10 times for a *Unique Files Benchmark.* The media server performance is 2.5-3 times higher under the *Single File Benchmark* than under the *Unique Files Benchmark.* This quantifies the performance benefits for multimedia applications when media streams are delivered from memory.

[0116] As described in co-pending U.S. Patent Application Number 10/306,279 entitled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER" and in the L. Cherkasova Paper, a set of basic benchmark measurements may be used to derive a cost function that defines a fraction of system resources needed to support a particular media stream depending on the stream bit rate and type of access (memory file access or disk file access), including the following costs:

A) $\cos t_{X_i}^{memory}$ - a value of cost function for a stream with memory access to a file encoded at $X_i\ Kb/s$; and

B) $\cos t_{X_i}^{disk}$ - a value of cost function for a stream with disk access to a file encoded at $X_i\ Kb/s$.

[0117] Many admission control strategies proposed in the existing literature use the "fixed" maximum bandwidth a server is capable of delivering as the main "scheduling" resource for admission of a new stream. Evidently, the amount of bandwidth a server is capable of delivering is variable and depends on the encoding bit rates of current streams in progress and the access type: memory versus disk.

[0118] One goal of an admission control mechanism is to prevent a media server from becoming overloaded. Overloading a media server typically results in violation of real-time properties of the media applications that it supports. Typically, an overloaded media server continues to serve all of the accepted streams but the quality of service degrades: the packets for accepted streams are sent with violation of "on-time delivery" and in such a way that the quality of the stream received by a client is compromised.

[0119] An embodiment of the present invention provides an admission controller for managing admission of requests for streaming media files to a media server in a manner that enables optimum utilization of the media server's resources while ensuring that desired QoS guarantees are maintained. In accordance with one embodiment, the admission controller performs two main procedures when evaluating whether a new request r_{new}^f can be accepted for service by a media server:

1) *Resource Availability Check.* During this procedure, a cost of a new request r_{new}^f is evaluated. To perform such evaluation, a current memory state of media server is computed using a segment-based access model in the manner described above. From the memory state, the admission controller can identify whether a prefix of requested file *f* is residing in memory, and thus determine whether request r_{new}^f will have a cost of accessing memory or disk correspondingly. Then, the admission controller checks whether, in the current time (at which the request r_{new}^f is received, the media server has enough available resources (capacity) to accommodate the resource requirements of such new request r_{new}^f; and

2) *QoS Guarantees Check.* During this procedure, the admission controller verifies that the acceptance of request r_{new}^{f} will not violate the QoS guarantees of already accepted requests at any point in the future, i.e., the media server will not enter an overloaded state at any point in the future.

[0120] Turning to FIGURE 9, an example operational flow diagram of one embodiment of an admission controller is shown. In operational block 901, the admission controller receives a new request r_{new}^{f} requesting service of a streaming media file from a media server. In operational block 902, the current memory state of the media server is determined (e.g., using the techniques described above for computing a current memory state). In operational block 903, a cost for the newly received request is determined. For instance, the type of access of the request (e.g., whether a memory access or a disk access) may be determined, and a corresponding $cost^{memory}$ or $cost^{disk}$ may be determined for the request. As described further herein, the determined memory state (of block 902) may be used in block 903 for determining the cost of r_{new}^{f}.

[0121] In operational block 904, the admission controller determines whether the media server has sufficient available resources to accommodate the newly received request r_{new}^{f}. That is, the computed cost of the newly received request is used to determined whether the media server's currently available resources are sufficient for servicing the request. If the media server does not have sufficient available resources to accommodate the newly received request, then operation advances to block 906 whereat the request is rejected (i.e., not admitted for service by the media server). Otherwise, if the media server has sufficient resources to accommodate the request, operation advances to block 905.

[0122] In operational block 905, the admission controller determines whether admission of the newly received request r_{new}^{f} for service by the media server will cause violation of a desired QoS of any of the already active requests being serviced by the media server. If admission of the request r_{new}^{f} will cause violation of a desired QoS of any of the already active requests, operation advances to block 906 whereat the request r_{new}^{f} is rejected (i.e., not admitted for service by the media server). Otherwise, if it is determined that admission of the request r_{new}^{f} will not cause a violation of the desired QoS for any of the

already active requests, operation advances to block 907 whereat request r_{new}^f is admitted for service by the media server.

[0123] In certain embodiments, a current memory state is computed for the media server in accordance with the techniques described above. From the memory state, the admission controller can identify, in response to a received request r_{new}^f, whether a requested file f or its initial prefix is residing in the media server's memory, and thus whether the request r_{new}^f will have a cost of access to memory or disk correspondingly. If the media server has enough currently available capacity to accommodate the resource requirements of new request r_{new}^f, as determined in block 904 of FIGURE 9, then request r_{new}^f may be conditionally accepted. Otherwise, request r_{new}^f is rejected.

[0124] When there is enough currently available server capacity to admit a new request r_{new}^f, an embodiment of the admission controller still ensures that the acceptance of request r_{new}^f will not violate the QoS guarantees of already accepted requests over their duration and that the media server will not enter an overloaded state at any point in the future, such as in block 905 of FIGURE 9. In particular, the admission controller assesses the following two situations:

1) Let a new request r_{new}^f be a disk access. In this case, there is a continuous stream of new, additional bytes transferred from disk to memory (amount of new bytes is defined by the file f encoding bit rate). This may result in replacement (eviction) of some "old" file segments in memory, such as some segments of file $\hat{f}$, for example. If there is an active request $r^{\hat{f}}$ which reads the corresponding file segments from memory and has a cost of memory access, then once the corresponding segments of file $\hat{f}$ are evicted (replaced) from memory, the request $r^{\hat{f}}$ will read the corresponding segments of file $\hat{f}$ from disk with increased cost of disk access. Accordingly, the admission controller assesses such situations whenever they may occur in the future for already accepted ("active") requests and evaluate whether the increased cost of impacted requests can be offset by the available capacity of the server at the corresponding time points; and

2) Let a new request r_{new}^f be a memory access. The admission controller assesses

whether the corresponding segments of file f may be evicted in some future time points by already accepted active disk requests, and whether the increased cost of request r_{new}^f can be offset by the available capacity of the server at the corresponding time points. That is, if request r_{new}^f is initially accepted as a memory access (e.g., the prefix of file f is available in memory at time T_{cur}), it is determined whether such request may (before it is fully serviced) become a disk access request and whether the media server will have sufficient resources to support such a disk access request.

[0125] Because certain embodiments of the admission controller attempt to ensure the QoS guarantees at any point in the future, an efficient procedure to advance the "system clock" and verify the QoS guarantees on a small set of future time points may be implemented in the manner described further below. It should be noted that an aim of the QoS validation process and related speculative computations of certain embodiments is to provide a determination of whether the request r_{new}^f can be accepted or not for service by the media server and ensure a desired QoS for all accepted requests.

[0126] FIGURES 10A-10B show an example operational flow diagram for QoS validation that is performed by an admission controller in accordance with one embodiment of the present invention. For instance, such QoS validation process may be performed in block 905 of FIGURE 9 in the overall operation of an embodiment of an admission controller. In accordance with the example embodiment for implementing QoS validation of FIGURES 10A-10B, in determining whether a received request r_{new}^f can be accepted for service by the media server, the admission controller identifies "safely replaceable bytes" in memory. More particularly, in the example embodiment of FIGURES 10A-10B, all of the active requests $ActReqs(T_{cur})$ are partitioned in two groups:

1) $ActMemReqs(T_{cur})$: active memory requests, i.e. the requests which have a memory access cost; and

2) $ActDiskReqs(T_{cur})$: active disk requests, i.e. the requests which have a disk access cost.

[0127] For instance, in block 1001, all $ActMemReqs(T_{cur})$ are identified, and in block 1002 all $ActDiskReqs(T_{cur})$ are identified. The requests from $ActMemReqs(T_{cur})$ access

their file segments in memory. Thus, they do not "bring" new bytes to memory - they only "refresh" the accessed file segments' timestamps with the current time. Only requests from $ActDiskReqs(T_{cur})$ bring new bytes from disk to memory and evict the corresponding amount of "oldest" bytes from memory.

[0128] There are simple dependencies between requests to the same file *f*. For instance, let $Reqs(T_{cur}) = \{r_1^f, r_2^f, \ldots, r_n^f\}$ be a sequence of requests accessing file *f* in interval $[T_{cur}^{mem}, T_{cur}]$ and let $t^{start}(r_1^f) \leq t^{start}(r_2^f) \leq \ldots \leq t^{start}(r_n^f)$, i.e. (r_1^f) is the oldest access and (r_n^f) is the most recent access to a file *f* in $[T_{cur}^{mem}; T_{cur}]$ interval. Let request $r_i^f\,(1 \leq i \leq n)$ be a memory access. Since request r_i^f reads the initial segment of file *f* from memory, and the most recent initial segment of file *f* in memory is written by the previous request r_{i-1}^f, there is the following dependency between accesses r_i^f and r_{i-1}^f. The offset $(t^{start}(r_i^f) - t^{start}(r_{i-1}^f))$ between the arrival times of requests r_{n-1}^f and r_n^f indicates the timestamp of the file segment in memory, which currently is accessible by request r_i^f:

$$T_{cur}^{act\_m}(r_n^f) = T_{cur} - (t^{start}(r_i^f) - t^{start}(r_{i-1}^f)).$$

[0129] As shown in operational block 1003, this example embodiment of the admission controller applies the above computation to each active memory request in $ActMemReqs(T_{cur})$:

$$T_{cur}^{act\_m} = \min_{r_i \varepsilon ActMemReqs(T_{cur})} T_{cur}^{act\_m}(r_i).$$

[0130] Time point $T_{cur}^{act\_m}$ indicates the timestamp of the "oldest-active" portion in memory, which is currently used (read) by some active memory requests. Any active memory request *r* such that $T_{cur}^{act\_m}(r) = T_{cur}^{act\_m}$ is referred to as an *active memory-oldest request.*

[0131] In operational block 1004, $TimeFileTable(T_{cur}^{mem}, T_{cur})$, which provides time ordering of file segments in memory, is used to identify all the file segments stored between T_{cur}^{mem} and $T_{cur}^{act\_m}$ and compute the corresponding amount of bytes in memory between these two time points as:

$$SafeReplBytes(T_{cur}) = UniqueBytes(T_{cur}^{mem}, T_{cur}^{act\_m}).$$

[0132] Thus, the computed *SafeReplBytes*(T_{cur}) are the bytes that can be safely replaced in memory (e.g., by new bytes being read from disk) without impacting any currently active memory requests. For instance, turning briefly to FIGURE 11, an example of such safely replaceable bytes that may be determined is shown.

[0133] Returning to FIGURES 10A-10B, this example embodiment of an admission controller next logically advances the system clock. That is, a logical system clock is advanced to enable evaluation of the media server's performance at some time in the future. For instance, the logical system clock may be advanced to a particular point in future to determine how the media server will manage its load at that point in future. Thus, the admission controller may be forward-looking to anticipate the state of the media server at point(s) in the future, and can thus evaluate how the media server's future state will be impacted by actions taken currently, such as admitting a newly received request r^f_{new}, as described further below.

[0134] As shown in operational block 1005, the admission controller may use the information about file encoding bit rates as well as the future termination times for requests $ActDiskReqs(T_{cur})$ to compute a time duration Δt during which the requests from $ActDiskReqs(T_{cur})$ will either transfer from disk to memory the amount of bytes equal to $SafeReplBytes(T_{cur})$ or all of them terminate. It is safe to advance clock to time $T_{cur} + \Delta t$ because during this time period the cost of accepted requests stays unchanged, and therefore the media server is guaranteed from entering the overloaded state in interval $[T_{cur}, T_{cur} + \Delta t]$. In order to make the QoS validation process terminate within a limited number of steps, this embodiment attempts to advance the clock at each step either beyond the closest termination event or beyond a designated $ClockAdvanceTime$. $ClockAdvanceTime$ is a parameter, which may be set depending, for example, on a workload presented to the media server. For instance, for a workload with long duration streams (e.g., long videos), $ClockAdvanceTime$ might be set to 1 minute or higher, while for a workload with short duration streams, $ClockAdvanceTime$ might be set, for example, to 5 seconds.

[0135] In operational block 1006, the admission controller determines if in interval $[T_{cur}, T_{cur} + \Delta t]$ all the requests from $ActDiskReqs(T_{cur})$ have reached their termination. If all of the requests from $ActDiskReqs(T_{cur})$ are determined in block 1006 to reach their

termination in interval $[T_{cur}, T_{cur} + \Delta t]$, operation advances to block 1008 whereat request r_{new}^{f} is admitted for service by the media server. Alternatively, operation advances to block 1007 and the admission controller determines if in interval $[T_{cur}, T_{cur} + \Delta t]$ all the requests from $ActMemReqs(T_{cur})$ have reached their termination. If all of the requests from $ActMemReqs(T_{cur})$ are determined in block 1007 to reach their termination in interval $[T_{cur}, T_{cur} + \Delta t]$, operation advances to block 1008 whereat request r_{new}^{f} is admitted for service by the media server. More specifically, if either of the conditions of blocks 1006 and 1007 are found to be satisfied, then the media server is guaranteed from entering an overloaded state in the case of request r_{new}^{f} being accepted, and thus such request is accepted for service in block 1008.

[0136] Otherwise, operation advances to block 1009 (via connector "A") shown in FIGURE 10B whereat the admission controller determines if $\Delta t \geq ClockAdvanceTime$. If $\Delta t \geq ClockAdvanceTime$, operation advances to block 1011, as described further below. Otherwise, operation advances to block 1010 whereat the admission controller determines whether there is one or more termination events within interval $[T_{cur}, T_{cur} + \Delta t]$. If one or more termination events are determined within interval $[T_{cur}, T_{cur} + \Delta t]$, operation advances to block 1011, which is described further below – otherwise, operation advances from block 1010 to block 1014.

[0137] Let $TermReqs(T_{cur}, T_{cur} + \Delta t)$ denote all the requests which terminate in interval $[T_{cur}, T_{cur} + \Delta t]$. In operational block 1011, the available server capacity is increased by $cost(r_i)$ for each request $r_i \in TermReqs(T_{cur}, \Delta T)$. Then, in operational block 1012, the admission controller computes an updated memory state and all of the corresponding data structures (e.g., the segment-based models) at time point $T_{cur} + \Delta t$. After that, the whole procedure is repeated with respect to $T_{cur} = T_{cur} + \Delta t$. Thus, in operational block 1013, time T_{cur} set to $T_{cur} + \Delta t$, and operation returns to block 1001 (via connector "B") to repeat the above-described processes with respect to $T_{cur} = T_{cur} + \Delta t$.

[0138] If $\Delta t \leq ClockAdvanceTime$ (in operational block 1009) and there is no termination events within interval $[T_{cur}, T_{cur} + \Delta t]$ (in operational block 1010), operation

advances to block 1014. Let $ActMemOldestReqs(T_{cur}) = \{r_i, \ldots, r_k\}$ be a set of all active memory-oldest requests, i.e. for any $r_i : T_{cur}^{act-m}(r_i) = T_{cur}^{act-m}$, $(1 \leq i \leq k)$. The active memory-oldest requests are accessing the file segments with the "oldest" timestamps in memory, and therefore, they are the most vulnerable to a situation that corresponding file segments can be evicted from memory. If this happens (i.e., their corresponding file segments are evicted from memory), these requests will incur the increased cost of disk access, and thus the admission controller attempts to identify those occurrences and ensure that there is enough available server capacity to offset this additional cost. Accordingly, in operational block 1014, for any $r_i \in ActMemOldestReqs(T_{cur})$ the admission controller assigns a corresponding cost of disk access (such process is referred to herein as a *request cost update*).

[0139] In operational block 1015, the admission controller determines whether there is sufficient available server capacity to offset this additional cost. If there is not enough available server capacity to offset the corresponding request cost updates, then the end of the QoS validation process is reached and request r_{new}^f is rejected in block 1016. If, on the other hand, there is enough available server capacity to offset the corresponding request cost updates, then requests $ActMemOldestReqs(T_{cur})$ are removed from $ActMemReqs(T_{cur})$ and are added to $ActDiskReqs(T_{cur})$ in operational block 1017. And, in operational block 1018, the available server capacity is correspondingly updated (e.g., decreased). After that, time point T_{cur}^{act-m} is recomputed with respect to an updated set of $ActMemReqs(T_{cur})$, and operation returns to block 1003 (via connector "C") shown in FIGURE 10A and the process continues for the updated data structures.

[0140] The QoS validation process described above in FIGURES 10A-10B guarantees to terminate and to provide the answer whether request r_{new}^f should be accepted or not.

[0141] In an embodiment of the admission controller, two types of events may be encountered by the controller: 1) *acceptance* of new requests, and 2) *termination* of previously accepted (active) requests. In these time points, the amount of available server resources is reevaluated by the controller. For instance, in the event of an acceptance of a new request, additional server resources may be allocated for servicing the new request (thus decreasing the

amount of available server resources), and in the event of termination of an active request, server resources allocated for servicing such terminated request may be released (thus increasing the amount of available server resources). It should be understood that multiple events may be encountered in the same time point. The following notations are used herein:

• *ActReqs(t)*: the set of requests that are currently in progress, i.e. "active" at time t ;

• *TermReqs(t)*: the set of requests that are suppose to terminate at event time t ;

• *Cap*: the absolute server capacity. In an example embodiment, the *Cap* is set to 1, and the *cost function* for requests is derived using this setting. For example, if a media server can support N concurrent disk accesses to files encoded at X_i Kb/s then the cost $\cos t_{X_i}^{disk}$ Of the disk request to a file encoded at X_i Kb/s is $\frac{1}{N}$; and

• $ACap(t)$: the available server capacity at time t. In the initial time point, one may use $ACap(T_{init}) = 0.9 \times Cap$.

[0142] An embodiment of the admission controller handles events as described below. Let the current event time (i.e., a time at which an event is encountered or detected by the admission controller) be T_{cur}. If there are events of both types: termination of some already accepted (active) requests and acceptance of a new request, then the termination event activities are performed first (the corresponding resources are released), and the acceptance event activities are performed after that. An example embodiment of the admission controller performs the actions described below depending on the event type.

[0143] First, let us consider the actions of an example embodiment of the admission controller for an acceptance of a new request event. To evaluate whether a new request r_{new}^f can be accepted at time T_{cur}, the admission controller performs the following two procedures: 1) Resource Availability Check and 2) QoS Guarantees Check, which are each described further below.

[0144] During the Resource Availability Check procedure, a cost of a new request r_{new}^f is evaluated via computation of a current memory state of the media server as described above. In particular, this example embodiment of the admission controller computes a timestamp T_{cur}^{mem} such that only file segments accessed at time T_{cur}^{mem} and later are residing in

memory. Additionally, for each request $r \in ActMemReqs(T_{cur})$, the admission controller computes timestamp $T_{cur}^{act\text{-}m}(r)$ corresponding to the oldest-active portion of memory which currently is accessible by the active memory requests. After that, the admission controller identifies if there are any requests $r \in ActMemReqs(T_{cur})$ such that $T_{cur}^{act\text{-}m}(r) \leq T_{cur}^{mem}$. This situation reflects that while request r has been originally a memory access, the current memory state has changed and request r at time T_{cur}^{mem} streams the corresponding bytes from disk. Hence, access r requires a corresponding *cost update*. Let *UpdReqs* denote all such requests (this set may be empty). Since the QoS validation process described above in conjunction with FIGURES 10A-10B guarantees that there is enough available server capacity in the corresponding time points, the admission controller runs a procedure of cost update for requests $r \in UpdReqs$: for any $r \in UpdReqs$, the admission controller assigns a corresponding cost of disk access, these requests are removed from $ActMemReqs(T_{cur})$ and are added to $ActDiskReqs(T_{cur})$, and the available server capacity is correspondingly decreased. Then, the admission controller checks whether the media server still has enough available resources (capacity) to accommodate the resource requirements of a new request r_{new}^{f}. In case of a positive outcome (i.e., the admission controller determines that the media server still has sufficient capacity to service the new request), the admission controller moves on to the QoS validation process, described below.

[0145] During the QoS Guarantees Check procedures, this example embodiment of the admission controller verifies that by accepting the new request r_{new}^{f} the media server will not enter an overloaded state at any point in the future, as described above in conjunction with FIGURES 10A-10B. If the outcome of this QoS validation process is positive (i.e., it is determined that the media server will not enter an overloaded state at any point in the future), then new request r_{new}^{f} is accepted and the following actions are performed:

- the available server capacity is decreased by the $cost\ (r_{new}^{f}): ACap(T_{cur}) = ACap(T_{cur}) - cost\ (r_{new}^{f})$;
- $ActReqs(T_{cur}) = ActReqs(T_{cur}) \bigcup r_{new}^{f}$; and
- let $T' = T_{cur} + duration\ (r_{new}^{f})$, then $TermReqs(T') = TermReqs(T') \bigcup r_{new}^{f}$.

[0146] Now let us consider the actions of an example embodiment of the admission controller for termination of currently accepted (active) requests. In the time points corresponding to termination events, the following actions are performed:

- $ActReqs(T_{cur}) = ActReqs(T_{cur}) \setminus TermReqs(T_{cur})$; and
- The server capacity is increased by the cost of the terminated requests:

$$ACap(T_{cur}) = ACap(T_{cur}) + \sum_{r \in TermReqs(T_{cur})} cost(r).$$

[0147] The above describes an example implementation of an admission controller in accordance with an embodiment of the present invention. Utilizing the techniques described above, a cost-aware admission controller may be implemented to manage the admission of requests to a streaming media server, which provides desirable performance benefits over traditional admission techniques for streaming media servers. In order to evaluate the performance benefits of such a cost-aware admission controller, we constructed a simulation model of the above-described example implementation of an admission controller in Java. Using this simulation model, we compared the example admission controller's performance against the traditional pessimistic strategy for a streaming media server. The pessimistic admission control strategy is based on measuring the media server capacity as a number of concurrent streams the disk subsystem is capable of supporting. When a new request arrives, it is assigned the cost of disk access. If there is enough available capacity to accommodate the resource requirements of a new request, it is accepted.

[0148] For workload generation, we used the publicly available, synthetic media workload generator known as *Medisyn*. We performed a sensitivity study, using two workloads closely imitating typical parameters of real enterprise media server workloads. The overall statistics for *Workload 1* and *Workload 2* used in the study are summarized in Table 1 below:

	Workload 1	**Workload 2**
Number of Files	800	800
Zipf α	1.34	1.22
Storage Requirement	41 GB	41 GB
Number of Requests	41,703	24,159

[0149] Table 1: Workload parameters used in simulation study.

[0150] Both synthetic workloads have the same media file duration distribution, which can be briefly summarized via six following classes: 20% of the files represent short streams (e.g., videos) having duration 0-2 minutes, 10% of the streams have duration of 2-5 minutes, 13% of the streams have duration of 5-10 minutes, 23% of the streams have duration of 10-30 minutes, 21% of the streams have duration of 30-60 minutes, and 13% of the streams have durations longer than 60 minutes.

[0151] The file bit rates are defined by the following discrete distribution: 5% of the files are encoded at 56Kb/s, 20% of the files are encoded at 112Kb/s, 50% of the files are encoded at 256Kb/s, 20% of the files are encoded at 350Kb/s, and 5% of the files are encoded at 500Kb/s. The file popularity is defined by a Zipf-like distribution with α shown in Table 1. In summary, *Workload 1* has a higher locality of references than *Workload 2*: 90% of the requests target 10% of the files in *Workload 1* compared to 90% of the requests targeting 20% of the files in *Workload 2*. Correspondingly, 10% of the most popular files in *Workload 1* have an overall combined size of 3.8GB, while 20% of the most popular files in *Workload 2* use 7.5GB of storage space.

[0152] Session arrivals are modeled by a Poisson process with rate corresponding to the media server overload. We performed a set of simulations for a media server with different memory sizes of 0.5GB, 1GB, 2GB, and 4GB. We defined the server capacity and the cost functions similar to those measured using the experimental testbed described above for deriving cost functions. In our experiment, we used $cost_{x_i}^{disk} / cost_{x_i}^{memory} = 3$, i.e. the cost of disk access is 3 times higher than the cost of the corresponding memory access to the same file.

[0153] The performance results for both workloads from our experiment are shown in FIGURES 12A (showing results for *Workload 1*) and 12B (showing results for *Workload 2*). They represent a normalized throughput improvement under the example cost-aware admission controller compared to the pessimistic admission control strategy using two metrics: 1) the number of accepted requests and 2) the total number of transferred bytes. As can be seen, the example cost-aware admission controller significantly outperforms the pessimistic strategy for both workloads while providing the hard QoS guarantees for media service in the manner described above. The media server's performance can be significantly improved via efficient memory management even for a media server with a relatively small memory size. Despite the fact that *Workload 2* has much less reference locality and its popular files occupy twice as much

space compared to that of *Workload 1* in our experiment, the performance improvements under the cost-aware admission controller for *Workload 2* are only slightly lower than for *Workload 1*.

[0154] Embodiments of the present invention may be utilized to model the memory of a media server. Such a memory model may be used, for example, in implementing an admission control policy for the media server for managing the acceptance of client requests to be serviced by the media server (e.g., to ensure optimal resource utilization and/or a desired level of quality of service). For instance, the memory of a media server may be modeled during its runtime, and such memory model may be utilized for measuring the capacity of the media server in supporting actual workloads applied thereto by a population of clients. For example, the capacity of the media server may be measured in accordance with the teachings of co-pending U.S. Patent Application Number 10/306,279 titled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER," using the memory model for determining the current memory state of the media server. Thus, when actually implemented in a client-server network, a media server's capacity (e.g., its available capacity) may be monitored for supporting actual workloads applied thereto using a derived cost function, as described in co-pending U.S. Patent Application Number 10/306,279 titled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER." This is particularly attractive in systems in which resources may be dynamically allocated, such as in Utility Data Centers (UDCs), for supporting the applied workloads.

[0155] It should be understood that while the example embodiments described above use the cost functions, as derived in co-pending U.S. Patent Application Number 10/306,279 entitled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER" and in the L. Cherkasova Paper, application of the present invention is not intended to be limited solely to those cost functions. Rather, in alternative embodiments the memory modeling approach described herein may be used for implementing other admission control strategies now known or later developed, including other cost-aware admission control strategies.

[0156] FIGURE 13 shows an example UDC 1300 in which resources may be dynamically allocated. Such a UDC 1300 may be used for implementing a media server complex in which resources are dynamically allocated for the media server responsive to the

workload applied thereto in accordance with the measured capacity of the media server. Implementations of UDC 1300 are known in the art and therefore UDC 1300 is only briefly described herein. As shown in FIGURE 13, UDC 1300 comprises data center management logic 1301 that is operable to manage the allocation of resources in UDC 1300. UDC 1300 is coupled to a communications network, such as the Internet 1302 and/or Intranets 1303, thus enabling access by clients (not shown) via such communication networks. Network virtualization logic 1304 and storage virtualization logic 1305 is also included. UDC 1300 further comprises cluster pool 1306, network-attached storage (NAS) pool 1307, load balancer pool 1308, firewall pool 1309, and storage pool 1310. Again, data center management logic 1301 is operable to manage the allocation of resources, such as resources available in cluster pool 1306, NAS pool 1307, and storage pool 1310. Thus, by modeling the current memory state of a media server in accordance with embodiments of the present invention and using such memory state for computing a cost function for measuring the capacity of the media server complex under an applied workload as described in co-pending U.S. Patent Application Number 10/306,279 titled "SYSTEM AND METHOD FOR MEASURING THE CAPACITY OF A STREAMING MEDIA SERVER," data center management logic 1301 may, responsive to the measured capacity, dynamically allocate the appropriate resources for supporting the applied workload.

[0157] When implemented via computer-executable instructions, various elements of embodiments of the present invention for modeling a media server's memory are in essence the software code defining the operations of such various elements. The executable instructions or software code may be obtained from a readable medium (e.g., a hard drive media, optical media, EPROM, EEPROM, tape media, cartridge media, flash memory, ROM, memory stick, and/or the like) or communicated via a data signal from a communication medium (e.g., the Internet). In fact, readable media can include any medium that can store or transfer information.

[0158] FIGURE 14 illustrates an example computer system 1400 adapted according to embodiments of the present invention. That is, computer system 1400 comprises an example system on which embodiments of the present invention may be implemented. Central processing unit (CPU) 1401 is coupled to system bus 1402. CPU 1401 may be any general purpose CPU. The present invention is not restricted by the architecture of CPU 1401 as long as CPU 1401 supports the inventive operations as described herein. CPU 1401 may execute the various logical instructions according to embodiments of the present invention. For example, CPU 1401 may execute machine-level instructions according to the exemplary operational flows

described above in conjunction with FIGURES 4, 7, 8, 9, 10A, and 10B. For instance, example computer system 1400 may comprise a media server for implementing the above-described operations of an embodiment of the present invention, or example computer system 1400 may comprise an admission controller that is included in a media server or is communicatively coupled to a media server for implementing the above-described operations of an embodiment of the present invention, as examples.

[0159] Computer system 1400 also preferably includes random access memory (RAM) 1403, which may be SRAM, DRAM, SDRAM, or the like. Computer system 1400 preferably includes read-only memory (ROM) 1404 which may be PROM, EPROM, EEPROM, or the like. RAM 1403 and ROM 1404 hold user and system data and programs, as is well known in the art.

[0160] Computer system 1400 also preferably includes input/output (I/O) adapter 1405, communications adapter 1411, user interface adapter 1408, and display adapter 1409. I/O adapter 1405, user interface adapter 1408, and/or communications adapter 1411 may, in certain embodiments, enable a user to interact with computer system 1400 in order to input information thereto.

[0161] I/O adapter 1405 preferably connects storage device(s) 1406, such as one or more of hard drive, compact disc (CD) drive, floppy disk drive, tape drive, etc. to computer system 1400. The storage devices may be utilized when RAM 1403 is insufficient for the memory requirements associated with storing data for application programs. Communications adapter 1411 is preferably adapted to couple computer system 1400 to network 1412 (e.g., network 103 of FIGURE 1).

[0162] User interface adapter 1408 couples user input devices, such as keyboard 1413, pointing device 1407, and microphone 1414 and/or output devices, such as speaker(s) 1415 to computer system 1400. Display adapter 1409 is driven by CPU 1401 to control the display on display device 1410.

[0163] It shall be appreciated that the present invention is not limited to the architecture of system 1400. For example, any suitable processor-based device may be utilized, including without limitation personal computers, laptop computers, computer workstations, and multi-processor servers. Moreover, embodiments of the present invention may be implemented

on application specific integrated circuits (ASICs) or very large scale integrated (VLSI) circuits. In fact, persons of ordinary skill in the art may utilize any number of suitable structures capable of executing logical operations according to the embodiments of the present invention.